UNIVERSIDAD DE ANTIOQUIA

Microservicio para identificar preferencias de clientes en los canales digitales de Bancolombia

Autor

Daniel Ortiz Montoya

Universidad de Antioquia

Facultad de Ingeniería, Departamento de Electrónica y Telecomunicaciones

Medellín, Colombia

2021

Microservicio para identificar preferencias de clientes en los canales digitales de Bancolombia.

Daniel Ortiz Montoya

Informe de práctica de:
Ingeniería de telecomunicaciones.

Asesores:

Erwin Alexánder Leal Piedrahita

Jorge Mario Cardona Ramirez

Universidad de Antioquia
Facultad de Ingeniería, Departamento de Electrónica y Telecomunicaciones
Medellín, Colombia
2021.

1. Resumen

Este proyecto surge debido a la necesidad de Bancolombia de detectar las funcionalidades más usadas por los usuarios de sus canales digitales. Para dar solución a dicho reto se desarrolló un microservicio que registra los eventos de los usuarios en los canales digitales. El microservicio cuantifica las veces que un usuario usa un servicio bancario en un canal digital. Para consultar las preferencias de los clientes el microservicio expone una API (Application Programming Interface), el API responde peticiones HTTP (Hypertext Transfer Protocol), lo que permite un consumo sencillo y ágil del servicio.

2. Introducción

Bancolombia es una entidad financiera Colombiana con sedes en Latinoamérica y Centroamérica. Para Bancolombia es una prioridad la satisfacción de sus clientes, por lo cual el desempeño de sus canales digitales es fundamental.

Con el objetivo de mejorar la velocidad, eficiencia de gasto y experiencia del cliente en los canales digitales, Bancolombia inició la construcción de un Middleware. El Middleware busca centralizar la gestión de los diferentes canales digitales de Bancolombia (aplicaciones móviles y aplicaciones web), en una sola capa usando una arquitectura de microservicios.

En la actualidad, Bancolombia requiere que el Middleware pueda detectar cuales son las funcionalidades más utilizadas por los usuarios en los canales digitales ("consultar saldo", "solicitar producto", "transferir dinero", entre otras). Al conocer las preferencias de los clientes, la plataforma podrá modificar la UI (User Interface - Interfaz de usuario) para lograr una mayor personalización de los canales digitales, y con ésto mejorar de manera considerable la UX (User eXperience - Experiencia de usuario). Por otro lado, esta información también podrá ser consumida por futuros agentes encargados de realizar la analítica de los canales digitales.

Para dar una solución a la problemática planteada, se desarrolló un microservicio en el middleware. El microservicio usa una BDOG (Base de Datos Orientada a Grafos) para registrar la información relacionada con la interacción de los usuarios en los canales digitales. Esta información permite determinar las funcionalidades más usadas por el usuario, así como el tipo de canal utilizado. Se espera que con el uso de la BDOG se pueda alcanzar un mejor control de la concurrencia, así como la eficiencia en el proceso de almacenar la información

El microservicio se desarrolló con Python 3.0, debido a que este lenguaje es soportado por varias de las herramientas usadas en el Middleware. Además, se trabajó con un bróker de mensajería el cual se encarga de la comunicación entre los microservicios. Para complementar el trabajo, también se desarrolló un componente encargado de simular los mensajes producidos por el Middleware. Finalmente, se usaron contenedores para ejecutar las instancias de la base de datos y el bróker de mensajería.

3. Objetivos

3.1. Objetivo general:

Desarrollar un microservicio encargado de identificar las funcionalidades más utilizadas por los clientes en los canales digitales de Bancolombia.

3.2. Objetivos específicos:

- Comprender la operación, arquitectura y funcionalidades de los canales digitales y el Middleware.
- Determinar las estrategias de despliegue y comunicación para el desarrollo del microservicio al interior del Middleware.
- Implementar un microservicio que utilice una BDOG para recolectar la información de las funcionalidades usadas por los clientes e identificar sus preferencias.
- Validar la correcta operación del microservicio.

4. Marco Teórico

4.1. Plataforma de gestión de los canales digitales en Bancolombia.

Bancolombia en su búsqueda por mejorar la velocidad, eficiencia de gasto y experiencia de cliente planteó la renovación de sus canales digitales. El proyecto de modernización de canales tiene como objetivo crear una plataforma que se encargue de gestionar los canales digitales.

La plataforma de gestión consta de 3 capas principales (ver Figura 1) Front, Middleware y API Manager (Capa de integración). Front es la capa que se encargará de la UI. El Middleware se encargará de gestionar las peticiones de la

capa Front y comunicarse con la capa de integración haciendo uso de microservicios. La capa de integración (API Manager), se encargará de gestionar las transacciones bancarias (Figura 1).

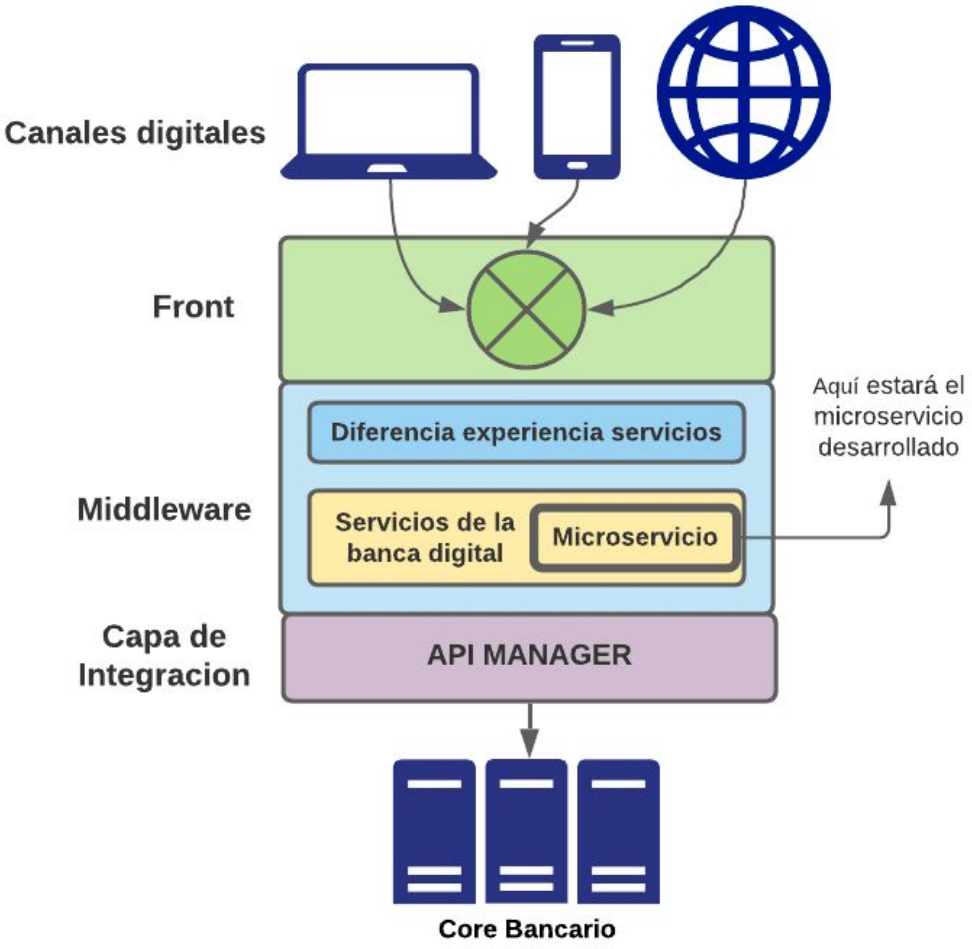


Figura 1. Plataforma de gestión de los canales digitales en Bancolombia.

La capa Middleware es una sección que contendrá servicios encargados de la seguridad y autentificación en los canales digitales, también microservicios encargados de manipular las operaciones bancarias. En esta capa se encuentra localizado el microservicio encargado de identificar las preferencias de clientes en los canales digitales, objetivo principal del proyecto.

La comunicación entre los microservicios se logrará gracias a un bróker de mensajería, donde llegarán todos los eventos como peticiones, transacciones, alertas, entre otros.

4.2. Arquitectura de microservicios

La arquitectura de microservicios es una arquitectura nativa de la nube cuyo objetivo es crear sistemas de software divididos en paquetes de pequeños servicios. Cada paquete tiene un comportamiento independiente y ejecuta sus propios procesos. Los paquetes se comunican a través de mecanismos ligeros, como una API (Application Programming Interface) u otros. Esto permite que cada microservicio pueda usar su propio lenguaje de programación y sus propios Frameworks. [2]

Migrar de las arquitecturas monolíticas (tradicionales) hacia una arquitectura de microservicios trae muchas ventajas. Algunas ventajas son: flexibilidad para adaptarse a las actualizaciones tecnológicas, reducción de los tiempos de comercialización (time-to-market), y una mejor estructuración del equipo de desarrollo en torno a los servicios [2].

Para ilustrar la arquitectura de microservicios se plantea un ejemplo[1], donde se tiene una aplicación de comercio electrónico que toma el pedido de los clientes, verifica inventario, verifica crédito y envía (ver Figura 2).

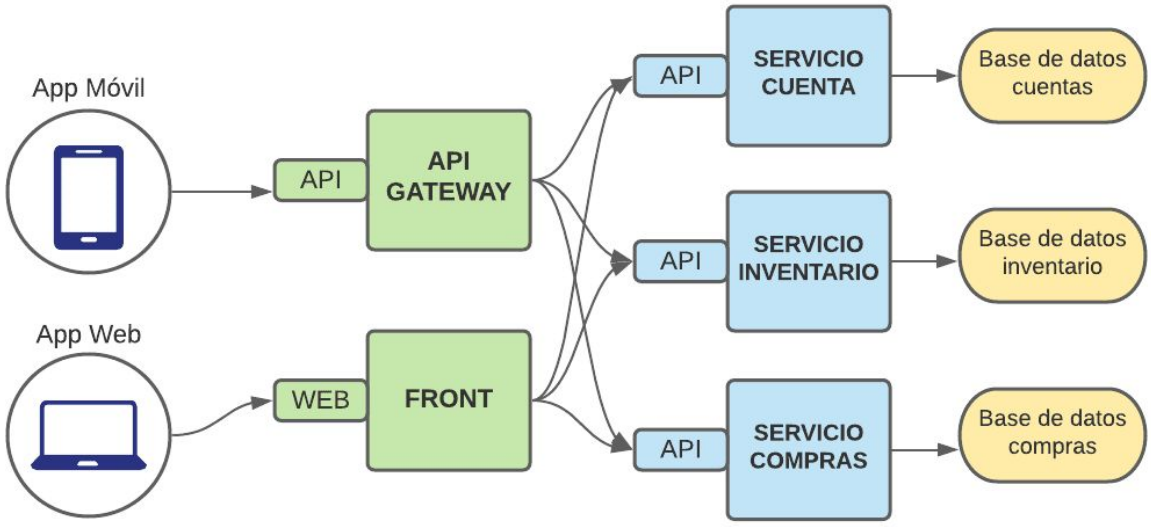


Figura 2. Ejemplo arquitectura de microservicio [1].

4.3. Bases de datos orientadas a grafos (BDOG)

El modelo de las bases de datos orientadas a grafos existe desde 1960. Este tipo de bases de datos han demostrado que proporcionan persistencia de manera constante, control de concurrencia y mecanismos de integración. Las bases de

datos relacionales tradicionales usan filas y columnas para almacenar la información, mientras que las BDOG almacenan la información en nodos y relaciones (ver Figura 3) [3].

Las características de las BDOG permiten que las entidades se encuentren conectadas entre sí, proporcionando una manera eficaz de almacenar información altamente relacionada. Debido a esta característica las BDOG se emplean en campos como la detección de fraude, sistemas de recomendación, bioinformática, inteligencia artificial, entre otros [3].

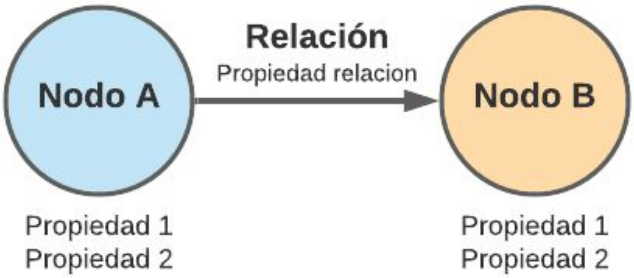


Figura 3. Esquema básico de una BDOG. [4]

En el proyecto se trabajó con una BDOG debido a que la información con la que se trabajo es altamente relacional.

El software de BDOG elegido para el desarrollo del proyecto es Neo4j. Neo4j es una BDOG nativa de código abierto que cuenta con una versión de escritorio y una imagen de Docker. Neo4j es la BDOG mas usada por la industria y en su sitio web cuenta con una guía muy completa para desarrolladores [4]. Además, existe una librería de Python llamada "py2neo" la cual facilita la gestión de Neo4j [7].

El lenguaje que usa Neo4j para ejecutar los comandos se llama Cypher. Cypher es un lenguaje declarativo de "Query" como lo es también SQL, permite al usuario almacenar y recuperar información de la base de datos. La sintaxis de Cypher proporciona una forma visual y lógica de hacer coincidir patrones de nodos y relaciones en el gráfico [4].

4.4. Contenedores (Docker)

Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de forma rápida y confiable en un entorno informático. Los contenedores aíslan el software de su entorno y garantizan que funcione de manera uniforme a pesar de las diferencias, por ejemplo, entre el desarrollo (construcción de la aplicación) y el despliegue (actividades necesarias para que una aplicación esté disponible para su uso) [5].

Docker es un proyecto de código abierto que gestiona el despliegue de aplicaciones dentro de contenedores de software. La tecnología Docker se centra en los requisitos de los desarrolladores y operadores de sistemas para separar las dependencias de las aplicaciones de la infraestructura.

Se trabajó con Docker durante el desarrollo del proyecto, siguiendo las recomendaciones del equipo de trabajo. Trabajar con Docker asegura que el software desarrollado opere en cualquier ambiente de trabajo. Además, facilita la distribución y escalamiento de la solución desarrollada.

4.5. Bróker de mensajería (RabbitMQ)

Un bróker de mensajería es una entidad encargada de aceptar y distribuir mensajes, similar a como lo hace una oficina de correos. RabbitMQ es un middleware que opera como bróker de mensajería, encargado de comunicar diversas aplicaciones o microservicios [6].

Se uso RabbitMQ ya que es el bróker de mensajería que designo Bancolombia para la comunicación entre los diferentes microservicios. RabbitMQ cuenta con una imagen de Docker. Además, desde un navegador web se puede acceder a un tablero donde muestra los detalles del bróker (ver Figura 7) [6].

Los componentes básicos de RabbitMQ son 3: El productor (P), el consumidor (C) y la cola. (ver Figura 4)

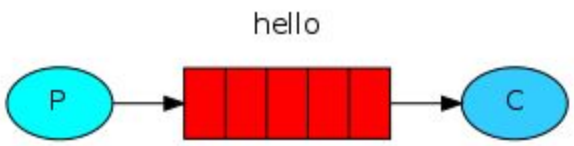


Figura 4. Componentes básicos RabbitMQ [6].

- Productor: Entidad que envía el mensaje [6].
- Cola: Una cola es el nombre de un buzón que vive dentro de RabbitMQ. Aunque los mensajes fluyen a través de RabbitMQ y sus aplicaciones, solo se pueden almacenar dentro de una cola. Una cola solo está limitada por los límites de memoria y disco del host, es esencialmente un gran búfer de mensajes. Muchos productores pueden enviar mensajes que van a una cola y muchos consumidores pueden intentar recibir datos de una cola [6].
- Consumidor: Entidad encargada de consumir el mensaje [6].

5. Estrategia de solución

5.1. Proceso de inducción y lineamientos del trabajo

Las primeras semanas se entró a un proceso donde se capacitaba en el modo de teletrabajo en Bancolombia. Se realizó una serie de cursos virtuales, sobre la cultura de trabajo en Bancolombia, además una reunión para conocer al equipo de trabajo. Después del proceso de adaptación se siguió con el proceso de contextualización. Durante el proceso de contextualización se realizó una reunión con los líderes del grupo de trabajo, en donde expusieron La plataforma de gestión en los canales digitales de Bancolombia.
Los líderes del proyecto plantearon el siguiente reto: "Evaluar estrategias y habilitadores tecnológicos que apalanquen la iniciativa de banco cognitivo en la distribución de canales digitales". La finalidad de este reto era que el estudiante implementara una estrategia de solución para enfrentar el reto planteado.

Después de realizar una pequeña investigación, se propusieron 2 posibles estrategias de solución para el reto planteado a los líderes del equipo. La estrategia de solución seleccionada y avalada por los líderes del equipo es la expuesta en este documento.

Como el microservicio va a estar dentro del Middleware (ver Figura 1), se debe cumplir con ciertos lineamientos (planteados por los líderes del equipo):

- El microservicio debe comunicarse con los otros microservicios a través de RabbitMQ (bróker de mensajería).

- La base de datos donde el microservicio va a almacenar la información debe estar desplegada en un contenedor de Docker.
- Para poder consumir el microservicio se debe crear un API que reciba peticiones HTTP y use el formato JSON (JavaScript Object Notation) para leer y enviar la información.

5.2. Solución

Después de familiarizarse con los conceptos relacionados con la problemática a abordar y encontrar las herramientas adecuadas para el desarrollo del proyecto y siguiendo las recomendaciones del equipo de trabajo, se procedió con la construcción del microservicio. La Figura 5 representa el diagrama de la solución planteada y desarrollada.

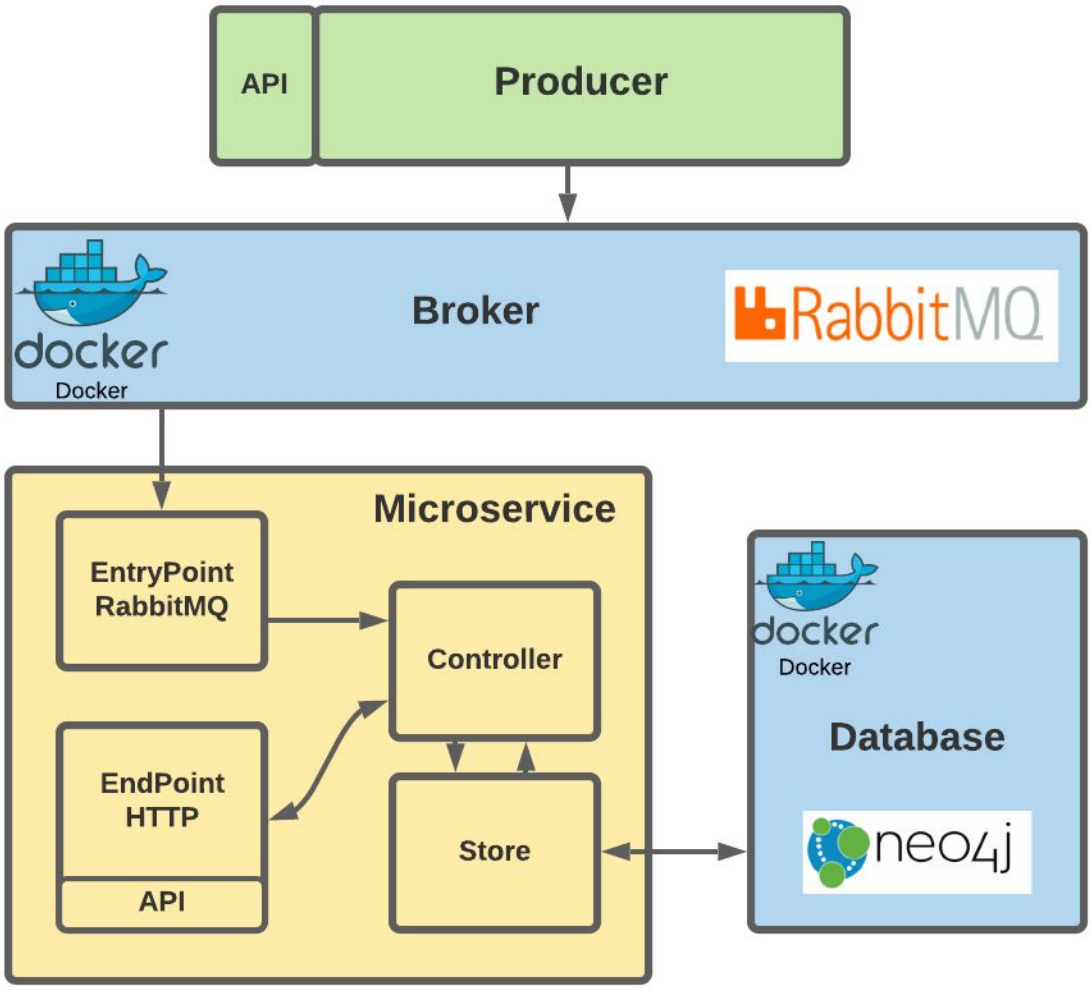


Figura 5. Diagrama de bloques de los componentes de la solución.

Cada vez que un cliente interactúa con una funcionalidad de los canales digitales, se genera un evento que llega como mensaje al bróker de mensajería. En el contexto de desarrollo de este proyecto el componente Producer se encarga de simular estos eventos y enviar los mensajes al bróker. El microservicio (Microservice) está constantemente "escuchando" al bróker, en espera de nuevos mensajes. Cada vez que entra un mensaje nuevo al microservicio, este se encarga de descomponerlo, extraer la información y registrarla en la base de datos. El microservicio también expone un API HTTP la cual retorna las funcionalidades más usadas por 'x' cliente. Por otro lado, el API también tiene la capacidad de retornar las cuentas bancarias a las que más se le transfiere.

A continuación, se describe de manera más detallada el desarrollo, funcionamiento y despliegue de cada uno de sus componentes.

5.2.1. Producer

El Middleware (Figura 1) se encuentra en desarrollo, por lo cual, por ahora no es posible poner en funcionamiento en el Middleware el microservicio desarrollado. Fue entonces necesario crear un microservicio temporal llamado Producer, encargado de simular los mensajes que llegan desde los otros microservicios al bróker de mensajería. Este microservicio genera los eventos haciendo uso de una API, bajo el protocolo HTTP, la cual envía en el body de la petición un JSON con la información que emulara una interacción generada por el cliente (transferencia, préstamo, recarga, etc.). El Producer pública la petición como un mensaje en RabbitMQ (ver Figura 6).

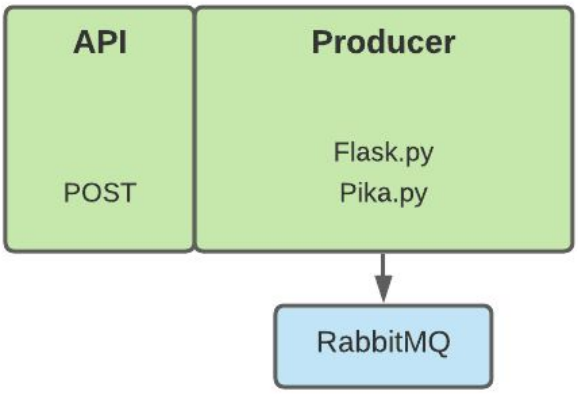


Figura 6. Diagrama de bloques para el componente Producer.

Las librerías 'Flask' y 'Pika' sirvieron de apoyo para la creación del Producer. 'Pika' como en el caso del EntryPoint RabbitMQ, es usado para gestionar la conexión, publicación y suscripción en RabbitMQ. Por otro lado, 'Flask' se encarga de crear el API (ver Código 1).

```
from flask import Flask, request, jsonify
import json
import pika
```

Código 1. Librerías Producer.

Cuando se genera un nuevo evento al API (trasferencia, préstamo, recarga, entre otros), por la ruta '/event', se ejecuta la función event(), ver Código 2. Esta función a su vez invoca la función 'send_rabbitmq', la cual se encarga de establecer la conexión con RabbitMQ y enviar el mensaje a una cola de RabbitMQ (ver Código 3).

```python
@app.route('/event', methods=['POST'])
def event():
    if request.method == "POST":
        req_Json = request.json
        req_message = json.dumps(req_Json)
        send_rabbitmq(req_message)
        return jsonify({"response":"event sent"})
```

Código 2. Función 'event'

```python
def send_rabbitmq(message):
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.exchange_declare(exchange='direct_logs', exchange_type='direct')
    key = "UserPreferencesServiceInput"

    channel.basic_publish(
        exchange='direct_logs', routing_key=key, body=message)
    print("[*][client] Event sent")

    connection.close()
```

Código 3. Función 'send_rabbitmq'.

5.2.2. Broker (RabbitMQ)

El bróker de eventos genera colas de mensajes, los microservicios del Middleware se suscriben y publican mensajes en dichas colas. En el proyecto se usó una 'routing_key' para discriminar los mensajes. RabbitMQ opera en el puerto TCP 5672 por defecto. Por otro lado, haciendo uso de un navegador web se puede acceder al puerto TCP 15672 para visualizar un tablero, este tablero nos brinda información relevante sobre el bróker (conexiones, colas, concurrencia, entre otras) (ver Figura 7).

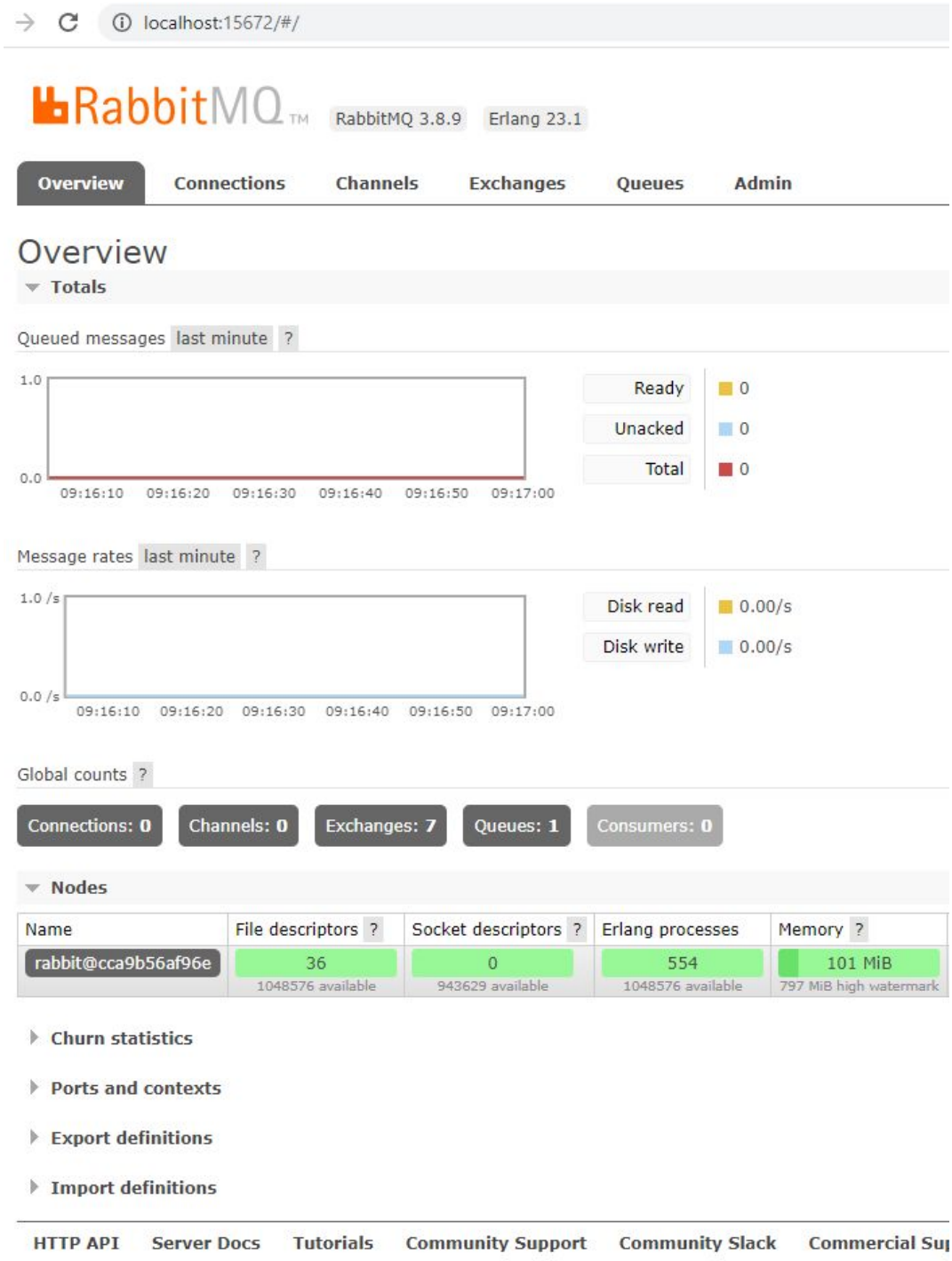


Figura 7. Tablero RabbitMQ.

5.2.3. Database (Neo4j)

5.2.3.1. Estructura de los grafos

Como se expuso anteriormente en el marco teórico, este microservicio utiliza una BDOG llamada Neo4j donde la información se almacena mediante una estructura de grafos. En este tipo de estructuras las entidades se representan por nodos (círculos) y relaciones (flechas), ver la Figura 8.

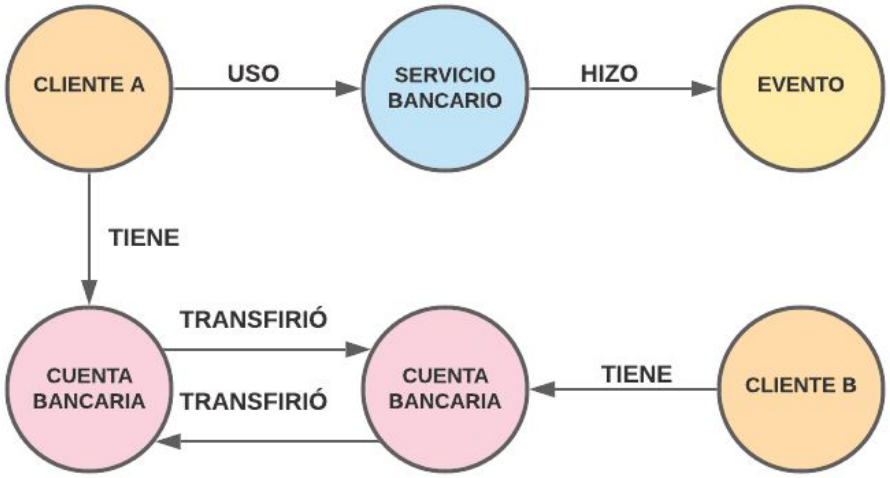


Figura 8. Estructura de la base de los grafos.

En la BDOG existen 4 tipos de Nodos:

- Cliente (Client): Representado por el número de identificación del cliente. Su única propiedad es 'name' la cual representa el número de identificación del cliente.

- Servicio Bancario (BankingService): Representado por el nombre del servicio bancario. Sus propiedades son: 'client' (ID del cliente que la uso), 'name' (nombre del servicio bancario) e 'importance' (parámetro para contabilizar el uso del servicio bancario).

- Cuenta Bancaria (BankingAccount): Representado por el número de la cuenta bancaria. Sus propiedades son: 'client' (ID del cliente dueño de la cuenta) y 'name' (número de la cuenta bancaria)

- Evento: Este nodo representa un evento creado por un cliente después de utilizar un servicio bancario. Los eventos están representados por un ID único del evento y sus propiedades varían según el servicio bancario que

utilizó el cliente. Por ejemplo, si el cliente realizó un préstamo (Loan), se crea un evento nuevo, las propiedades de este evento serán información del préstamo (valor del préstamo, fecha, tipo de préstamo, fecha limite, entre otras). Cada vez que el cliente usa un servicio bancario se crea un nodo Evento nuevo.

En La BDOG existen 4 tipos de Relaciones:

- USO (USE): Relaciona un nodo Cliente con un nodo Servicio Bancario. Se crea cuando un cliente consume un servicio bancario. Esta relación no tiene propiedades.

- HIZO (DID): Relaciona un nodo Servicio Bancario con un nodo Evento. Se crea cuando un cliente consume un servicio bancario. Esta relación no tiene propiedades.

- TIENE (HAVE): Relaciona un nodo Cliente con un nodo Cuenta Bancaria. Se crea cuando un cliente usa una cuenta bancaria para realizar una transferencia. Esta relación no tiene propiedades.

- TRANSFIRIÓ (TRANSFERRED): Relaciona un nodo Cuenta Bancaria con otro nodo Cuenta Bancaria. Se crea cuando una cuenta bancaria transfiere dinero a otra. Sus propiedades son: 'dstAccountNumber' (numero de la cuenta destino), 'srcAccountNumber' (numero de la cuenta origen), e 'importance' (parámetro para contabilizar las veces que se le trasfiere dinero a la cuenta destino desde la cuenta origen).

5.2.3.2. Creando grafos

A continuación, se muestran algunos comandos utilizados para crear el esquema de la base de datos usando Cypher. Se hace uso del browser de Neo4j para ejecutar y visualizar los resultados.

- Crear nodo cliente

- Comando de creación :

  "CREATE (c:Client {name:'Juan29', clientId:'0123', typeId:'CC'})"

- Comando de consulta :

  “ MATCH (c:Client {clientId:'0123'}) RETURN c ”

- Resultado (ver Figura 9):

Figura 9. Nodo cliente.

El comando de creación empieza con la palabra clave ‘CREATE’, ingresa el tipo del nodo, en este caso es tipo Client, seguido de los parámetros. El comando de consulta empieza con la palabra clave ‘MATCH o MERGE’, seguido de los datos necesarios para filtrar la consulta, seguido de la palabra clave ‘RETURN’ para retornar las entidades que cumplen con los parámetros. En este caso se usó el tipo de dato (Client) y un identificador único (clientId) para realizar la consulta.

- Crear nodo BankigService

- Comando de creación:

  “CREATE (b:BankingService {name:'Loan', clientId:'0123', importance:0})”

- Comando de consulta:

  “MATCH (b:BankingService {name:'Loan', clientId:'0123'}) RETURN b”

- Resultado (ver Figura 10):

Figura 10. Nodo BankingService

- Crear relación Client -> BankingService

- Comando de creación:

  MATCH (c:Client {clientId:'0123'})

  MATCH (b:BankingService {name:'Loan', clientId:'0123'})

  CREATE (c)-[:USE{weight_relationship:[0]}]->(b)

- Comando de consulta:

  MATCH (c:Client {clientId:'0123'})

  MATCH (b:BankingService {name:'Loan', clientId:'0123'})

  RETURN c,b

- Resultado(ver Figura 11):

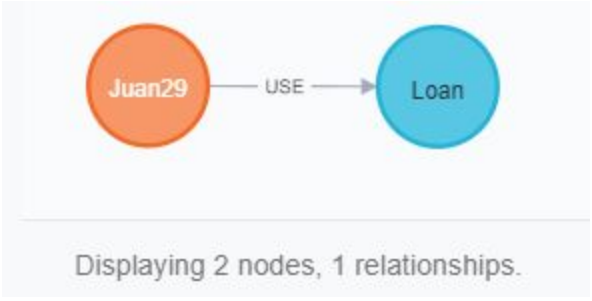


Figura 11. Relación: Cliente -> Servicio Bancario.

Es necesario anotar que los comandos expuestos son algunos de los que se utilizaron en el proyecto. Esto es solo una pequeña parte de la capacidad que

tiene Cypher en conjunto con Neo4j. Además, hay que tener en cuenta que se pueden obtener los mismos resultados usando otros comandos distintos.

5.2.3.3. Base de datos poblada (Ejemplo)

Con el fin de visualizar la base de datos poblada, se simularon unos pocos eventos, donde los Clientes con identificaciones "001" y "002" realizaron algunos préstamos y transferencias en los canales digitales. En la Figura 12. se expone el resultado obtenido usando el browser de Neo4j, el cual permite visualizar de manera gráfica la estructura de grafos obtenida.

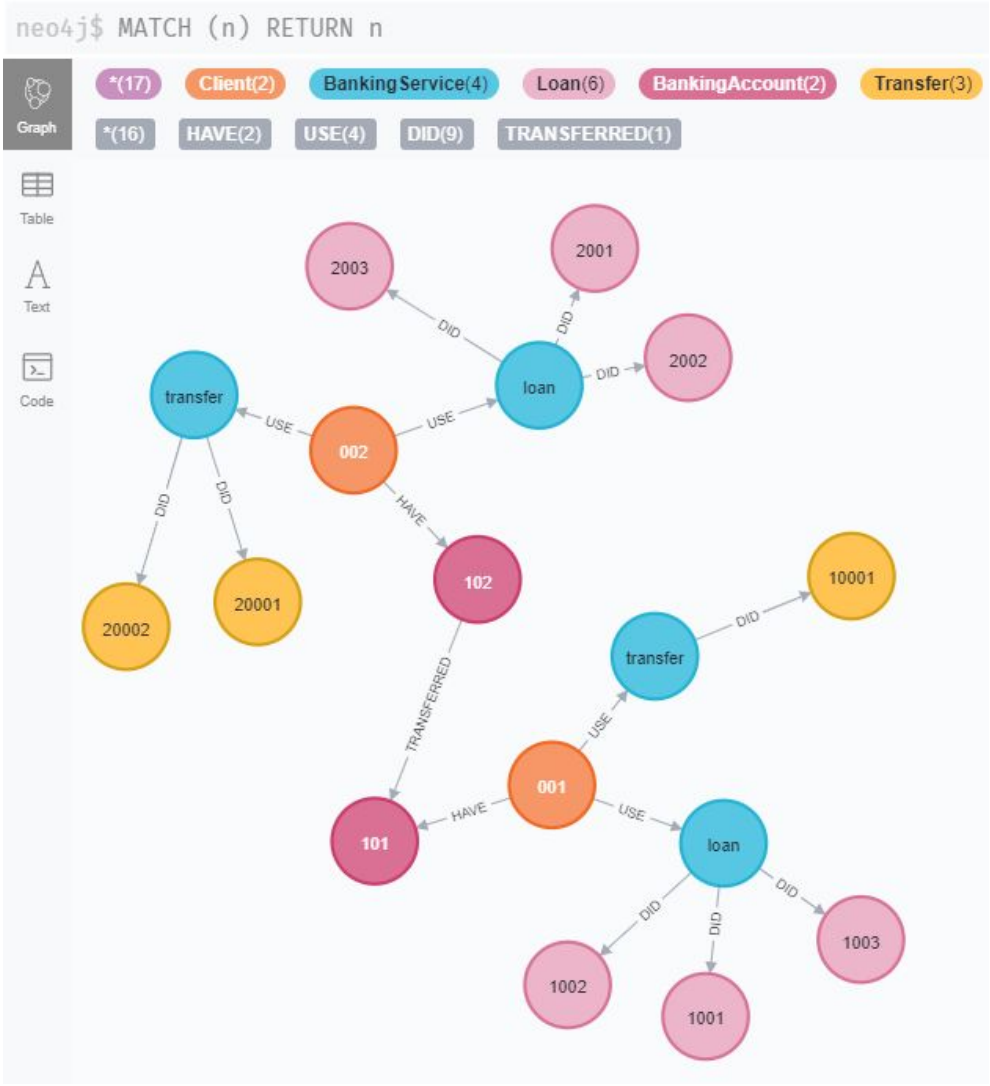


Figura 12. Base de datos poblada.

En la parte superior de la Figura 12. se pueden observar las entidades como Cliente, Servicio bancario, Cuenta bancaria, transferencia y Préstamo. Cada entidad se representa con un nodo y se diferencian entre ellos por el color.

También se puede observar cómo se relacionan entre sí (HAVE, DID, USE, TRANSFERRED).

5.2.4 Microservice

Microservice es el nombre del componente encargado de operar como el microservicio encargado de detectar las preferencias de los clientes. El microservicio requería una conexión a una base de datos (Neo4j) Y a un bróker de mensajería (RabbitMQ). Para restar complejidad a la solución lo más factible era desarrollar todo el Microservice en un solo lenguaje. Los lenguajes candidatos eran Java, JavaScript y Python, ya que dichos lenguajes son soportados por Neo4j y RabbitMQ. Debido a la experiencia adquirida durante la estadía en la universidad, además de una mayor documentación y sencillez, se decidió desarrollar la solución con Python 3.

El Microservice está compuesto por 4 componentes: EntryPoint RabbitMQ, EndPoint HTTP, Controller y Store (ver Figura 13). Al crear 4 componentes se logra tener una solución con un código más limpio y mejor estructurado.

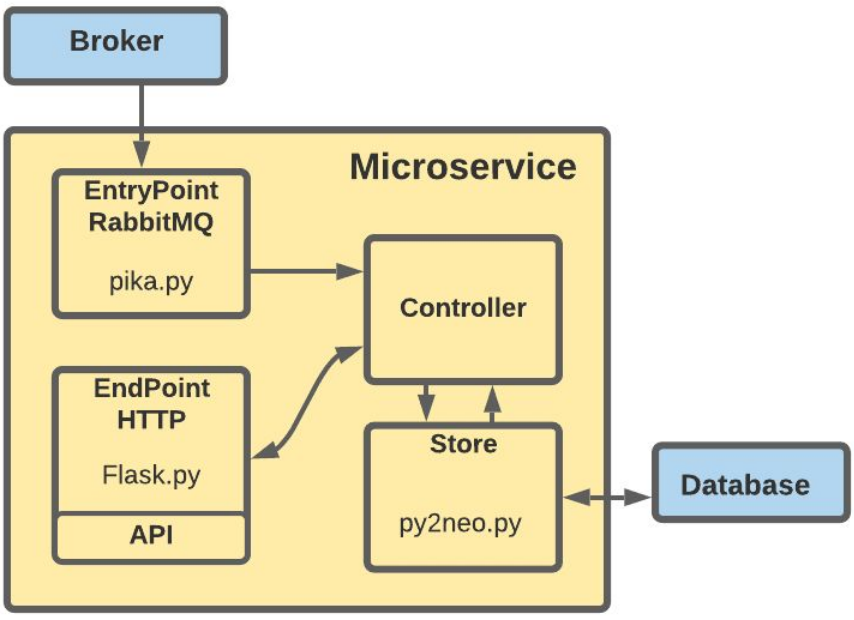


Figura 13. Diagrama de bloques del componente Microservice.

5.2.4.1 Store

El componente Store se encarga de realizar la conexión con la base de datos haciendo uso de la librería "py2neo". Py2neo cuenta con muchas funciones que facilitan la conexión, lectura y escritura en Neo4j (ver Código 4).

```
from py2neo import *
```

Código 4. Librerías Store

- Conexión con Neo4j:

  La función 'connect_to_neo4j_db' se encarga de ejecutar la función 'Graph()' de la librería Py2neo.'Graph()' se conecta a una instancia de la base de datos, y tiene varios parámetros (auth, host, port, password, scheme, secure, user, user_agent, max_connections). Para el desarrollo del Microservice se dejan los valores por defecto y solo se ingresa la contraseña. (ver Código 5)

```
def connect_to_neo4j_db(password):
    db = Graph(password=password)
    return db
```

  Código 5. Conexión con Neo4j.

- Creación de Nodos:

  Para crear los nodos se requiere una instancia de la base de datos, la cual es retornada por la función 'Graph' mencionada anteriormente. Py2neo cuenta con varias opciones para crear los nodos, la utilizada en el proyecto es la '.run'. Esta función ejecuta un comando Cypher directamente en la base de datos. Para ilustrar esto se expone una función implementada en el Código 6.

```
def create_client_node(db,clientId):
    command = "CREATE (client:Client {name:'%(name)s'} )" %{"name":clientId}
    db.run(command)
    print("[*][store] Client node name: {} , was created".format(clientId))
```

  Código 6. Creación de nodos.

  De manera análoga se crearon todos los otros nodos en la base de datos, como son BankingService, BankingAccount, Loan, entre otros.

- Creación de relaciones:

  De manera análoga a la creación de nodos, también se necesita una instancia de la base de datos y se usa la función '.run' para ejecutar el comando Cypher. Para ilustrar esto se expone una función implementada en el Código 7.

```
def create_client_account_relationship(db,clientId,accountNumber):
    command = """MATCH (c:Client {name:'%(clientName)s'})
MATCH (a:BankingAccount {name:%(accountNumber)s, client:'%(clientName)s'})
CREATE (c)-[:HAVE{graph_weight:[0]}]->(a)
""" %{"clientName":clientId, "accountNumber":accountNumber}
    db.run(command)
    print("[*][store] Relationship, Account. ({})-HAVE->({}) , was created"
          .format(clientId,accountNumber))
```

  Código 7. Creación relaciones.

  De manera análoga se crearon todas las relaciones en la base de datos.

- Añadir importancia

  La función encargada de añadir la importancia a un servicio bancario consta de dos pasos (ver Código 8 y 9)

  1. Se ejecuta un comando encargado de leer la importancia actual del servicio bancario, y se adiciona 1:

```
def add_importance_banking_service(db,clientId,bankingService):
    command_read = "MERGE (bs:BankingService {name: '%(bankingService)s', client:'%(name)s'}) return bs" %{
                                                                    "bankingService":bankingService,
                                                                    "name":clientId}

    query=db.run(command_read)
    results = [record for record in query.data()]
    importance = int(results[0]['bs'].get("importance"))
    importance = str(importance + 1)
```

  Código 8. Leer importancia.

  2. Se ejecuta un comando encargado de sobrescribir la información en la base de datos:

```
command_write="MERGE(bs:BankingService{name:'%(bankingService)s',client:'%(name)s'}) SET bs.importance=%(importance)s" %{
                                                                    "bankingService":bankingService,
                                                                    "name":clientId,
                                                                    "importance":importance}
db.run(command_write)
print("[*][store] Importance is {} in {}'s {} ".format(importance,clientId,bankingService))
```

Código 9. Escribir importancia.

De manera similar también se hace cuando se le añade importancia a una cuenta bancaria.

- Consultar preferencias:

  La función encargada de consultar las preferencias ejecuta un comando que retorna en orden descendente de importancia los servicios bancarios usados por el cliente. La función almacena esta información en una lista y la retorna (ver Código 10).

```
def more_important_banking_services(db, clientId):
    command = "MATCH (bs:BankingService{client:'%(name)s'}) RETURN bs ORDER BY bs.importance DESC" %{
        "name": clientId}
    query=db.run(command)
    results = [record for record in query.data()]
    preferences = []

    for i in range(len(results)):
        preferences.append(str(results[i]['bs'].get("name")))

    return preferences
```

Código 10. Consultar preferencias.

Utilizando una estructura de código similar se pueden determinar las cuentas bancarias "preferidas".

5.2.4.2 Controller

El Controller opera como un intermediario entre los componentes EntryPoint, EndPoint y la base de datos (Store). El controller analiza los mensajes que llegan, los descompone y toma las decisiones correspondientes. Además, el controller ejecuta las funciones del Store. Las librerías o módulos empleados en este componente son 'json' y 'store' (ver Código 11).

```
import store
import json
```

Código 11. Librerías Controller.

La comunicación en el Middleware, de forma similar a la mayor parte de aplicaciones de software, utiliza el formato JSON.

Los mensajes son las entidades que contienen la información del evento generado por el cliente en los canales digitales. En la Figura 14 se expone la estructura JSON de un mensaje generado por los microservicios del Middleware.

Figura 14. Estructura del mensaje ejemplo de un evento 'Loan' generado por un cliente.

Debido a que el Middleware todavía se encuentra en desarrollo, no está definido el formato de los mensajes, sin embargo, el mensaje de la Figura 14 será muy similar al empleado en producción. El mensaje de la Figura 14 representa un evento generado por un cliente en los canales digitales, después de que este realizara un préstamo (Loan). El ejemplo de la Figura 14 será empleado para explicar un flujo en el Controller.

- Flujo Controller (Evento 'Loan')

  Cuando llega un nuevo mensaje del EntryPoint RabbitMQ al Controller, la función 'client_event' se encarga de gestionarlo (ver Código 12).

```python
def client_event(password, body):
    db = store.connect_to_neo4j_db(password)
    eventJson = json.loads(body)

    clientId=eventJson["data"][0]["customerInformation"]["documentNumber"]
    bankingService = eventJson["type"]

    if (store.client_exists(db,clientId)):
        print("[*][controller] client exists")
    else:
        store.create_client_node(db,clientId)

    if (store.banking_service_exists(db,clientId,bankingService)):
        store.add_importance_banking_service(db,clientId,bankingService)
    else:
        store.create_banking_service(db,clientId,bankingService)
        store.create_client_banking_relationship(db,clientId,bankingService)

    according_banking_service(db, eventJson)

    message = json.dumps({
        "response": "registered",
    })

    return message
```

Código 12. Función 'client_event'.

La función 'according_banking_service' se encarga de detectar la funcionalidad usada por el cliente, y con esto ejecuta la función correspondiente a dicha funcionalidad (ver Código 13).

```python
def according_banking_service(password, eventJson):
    bankingService = eventJson["type"]

    if(bankingService == "loan"):
        write_loan_data(password, eventJson)
```

Código 13. Función 'according_banking_service'

Se ejecuta la función del 'Store' encargada de ingresar los datos en la base de datos (ver Código 14).

```
def write_loan_data(db, data):
    store.create_loan(db,
                      data["data"][0]["customerInformation"]["documentNumber"],
                      data["type"],
                      data["data"][0]["LoanInformation"]["loanNumber"],
                      data["data"][0]["LoanInformation"]["participationNumber"],
                      data["data"][0]["LoanInformation"]["paymentDate"],
                      data["data"][0]["LoanInformation"]["paymentType"],
                      data["data"][0]["LoanInformation"]["accountType"],
                      data["data"][0]["LoanInformation"]["accountNumber"],
                      data["data"][0]["LoanInformation"]["PaymentValue"],
                      data["data"][0]["LoanInformation"]["depositTransactionCode"],
                      data["data"][0]["LoanInformation"]["transactionDescriptionInDeposits"],
                      data["data"][0]["LoanInformation"]["transactionTrackingNumber"],
                      )
```

Código 14. Función 'write_loan_data'

El flujo para registrar eventos de los otros servicios bancarios en la base de datos es muy similar al expuesto anteriormente.

También se puede dar el caso de que el EndPoint HTTP envíe una petición para consultar las preferencias del cliente. En este caso el 'Json' es el que se observa en la Figura 15.

Figura 15. Estructura del mensaje ejemplo de una petición para obtener preferencias.

En la Figura 15. podemos observar un JSON usado para realizar la petición (desde el API del EndPoint HTTP) encargada de consultar las preferencias bancarias. Se usará este ejemplo para exponer un flujo del 'Controller'.

- Flujo controller (petición preferencias bancarias)

  La función 'client_banking_services_preferences' es invocada por el EndPoint. La función se encarga de retornar las 3 funcionalidades preferidas del cliente, esto en JSON (ver Código 15).

```
def client_banking_services_preferences(password, body):
    eventJson = json.loads(body)

    clientId=eventJson["data"][0]["customerInformation"]["documentNumber"]
    action = "response"
    first_preference = ""
    second_preference = ""
    third_preference = ""
    db = store.connect_to_neo4j_db(password)
    preferences = store.more_important_banking_services(db, clientId)

    try:
        third_preference = preferences[2]
    except IndexError:
        pass

    try:
        second_preference = preferences[1]
    except IndexError:
        pass

    try:
        first_preference = preferences[0]
    except IndexError:
        pass

    message = json.dumps({
        "action": action,
        "clientId": clientId,
        "pref_1": first_preference,
        "pref_2": second_preference,
        "pref_3": third_preference,
    })

    return message
```

Código 15. Función 'client_banking_services_preferences'.

El flujo es similar para retornar las cuentas a las que más se le transfiere.

5.2.4.3. EntryPoint RabbitMQ

Este componente se encarga de gestionar la conexión con el bróker de mensajería. Se usó la librería 'pika', esta librería dispone de funcionalidades que facilitan el trabajo con RabbitMQ (ver Código 16).

```
import pika
import controller as ct
import sys
import json
```

Código 16. Librerías EntryPoint RabbitMQ.

La función principal del componente es 'connect_rabbitmq'. Al invocar esta función, el componente se suscribe a una cola de eventos y este queda "escuchando", esperando por el ingreso de nuevos mensajes a dicha cola (ver Código 17).

```
def connect_rabbitmq():
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.exchange_declare(exchange='direct_logs', exchange_type='direct')

    result = channel.queue_declare(queue='', exclusive=True)
    queue_name = result.method.queue

    key = "UserPreferencesServiceInput"
    channel.queue_bind(
        exchange='direct_logs', queue=queue_name, routing_key=key)

    print(' [*][ep rabbit] Waiting for logs. To exit press CTRL+C')

    channel.basic_consume(
        queue=queue_name, on_message_callback=callback_new_event, auto_ack=True)

    channel.start_consuming()
```

Código 17. Función 'connect_rabbitmq'

Cuando el componente detecta un nuevo mensaje, se ejecuta la función 'callback'. La función 'callback' ejecuta la función del 'Controller' encargada de registrar un nuevo evento (ver Código 18).

```
def callback_new_event(ch, method, properties, body):
    ct.client_event(PASSWORD_DB,body)
```

Código 18. Función callback.

5.2.4.4. EndPoint HTTP

Este componente se encarga de gestionar las peticiones que consultan las preferencias. Se trabajó con 'Flask' de python. 'Flask' es un framework que permite crear aplicaciones web de manera corta y sencilla, se usó 'Flask' para crear un API (ver Código 19).

```
from flask import Flask, request, jsonify, abort
import json
import controller as ct
import sys
```

Código 19. Librerías EndPoint HTTP.

El EndPoint HTTP opera en el puerto TCP 9091, y es necesario que se ingrese la contraseña de la base de datos como parámetro externo para ejecutarlo (ver Códigos 20 y 21).

```
if __name__ == '__main__':
    read_password_db()
    app.run(debug=True, port=9091)
```

Código 20. Parametrización del puerto.

```
def read_password_db():
    global PASSWORD_DB

    try:
        PASSWORD_DB = sys.argv[1]
    except IndexError:
        print("[x][ep http] Enter the database password")
        exit()
```

Código 21. Parametrización contraseña.

Cuando se realiza una petición 'GET' con la ruta '/preferences/services', se ejecuta la función del Controller encargada de retornar los servicios bancarios más usados por el cliente (ver Código 22).

```python
@app.route('/preferences/services', methods=['GET'])
def preferences_services():
    if request.method == "GET":
        req_json = request.json
        req_message = json.dumps(req_json)

        try:
            res_message=ct.client_banking_services_preferences(PASSWORD_DB, req_message)
            res_json = json.loads(res_message)
            return jsonify(res_json)
        except:
            return abort(400)
```

Código 22. Ruta petición preferencias.

El proceso para consultar las cuentas bancarias "preferidas", es análogo al proceso mencionado anteriormente.

5.2.5. Dockerización

Como se observa en la Figura 5 (componentes en azul), los componentes Broker y Database se encuentran dockerizados.

- Base de datos (Database)

  Para desplegar la base de datos en Docker, lo primero es conseguir la imagen oficial de neo4j, con el comando que se expone en el Código 23.

  ```
  (base) C:\Users\PC>docker pull neo4j
  ```

  Código 23. Comando para traer imagen neo4j.

  Para correr la imagen ver el Código 24.

  ```
  (base) C:\Users\PC>docker run --name testneo4j -p 7474:7474 -p 7687:7687 -d --env
  NEO4J_AUTH=neo4j/test neo4j:latest
  ```

  Código 24. Comando para correr la imagen neo4j.

  Como se puede observar, se declara los puertos, el usuario, la contraseña y la versión.

- Bróker de eventos (Broker)

  Para desplegar la base de datos en Docker, lo primero es conseguir la imagen oficial de RabbitMQ, con el comando expuesto en el Código 25.

```
(base) C:\Users\PC>docker pull rabbitmq
```

Código 25. Comando para traer la imagen rabbitmq

Para correr la imagen ver el Código 26

```
docker run -d -p 15672:15672 -p 5672:5672 --name rabbitmq rabbitmq:3-management
```

Código 26. Comando para correr la imagen rabbitmq

Como se puede observar, se declara los puertos, el usuario, la contraseña y la versión.

Con el comando 'docker ps -a' se puede observar el estado de los contenedores.

6. Resultados y análisis

6.1. Demostración

Para exponer los resultados se procedió a realizar un flujo de trabajo completo en un ambiente local. En este flujo interactúan todos los componentes de la solución (ver Figura 16). Hay que tener en cuenta que se trata de un ambiente de desarrollo, por lo cual no se cuenta con muchos requerimientos de seguridad necesarios para un despliegue en producción.

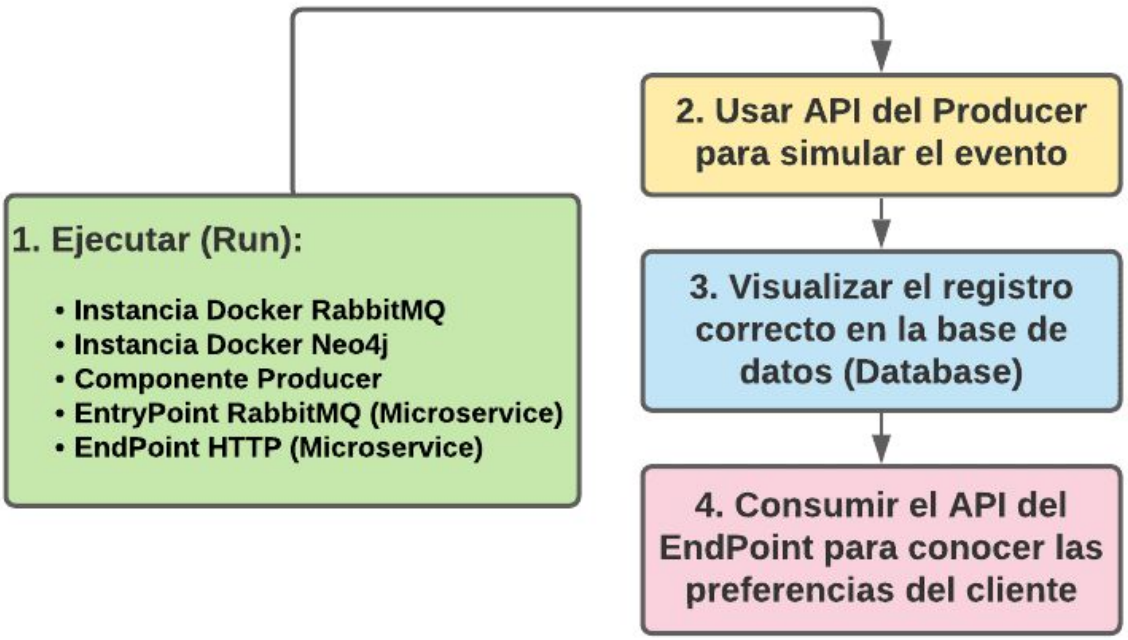


Figura 16. Diagrama de la demostración implementada.

A continuación, se describe a detalle el proceso para realizar la demostración.

6.1.1. Se ejecuta la instancia de RabbitMQ en Docker. Además, se ejecuta el componente 'Producer', 'EntryPoint RabbitMQ' y 'EndPoint RabbitMQ' del Microservice:

- Ejecución instancia rabbitmq (ver Código 27).

```
(base) C:\Users\PC>docker start rabbitmq
rabbitmq
```

Código 27. Ejecución instancia rabbitmq

- Ejecución componente Producer (Código 28).

```
(base) C:\Users\PC\code\proyecto_banco\proyecto\producer>python producer.py
 * Serving Flask app "producer" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 284-348-914
 * Running on http://127.0.0.1:9090/ (Press CTRL+C to quit)
```

Código 28. Ejecución componente 'Producer'.

- Ejecución 'EntryPoint RabbitMQ' del Microservice (Código 29).

```
(base) C:\Users\PC\code\proyecto_banco\proyecto\micro>python entry_point_rabbitmq.py 1234
 [*][ep rabbit] Waiting for logs. To exit press CTRL+C
```

Código 29. Ejecución 'EntryPoint RabbitMQ' del Microservice.

- Ejecución 'EndPoint HTTP' del Microservice (Código 30).

```
(base) C:\Users\PC\code\proyecto_banco\proyecto\micro>python end_point_http.py 1234
 * Serving Flask app "end_point_http" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 284-348-914
 * Running on http://127.0.0.1:9091/ (Press CTRL+C to quit)
```

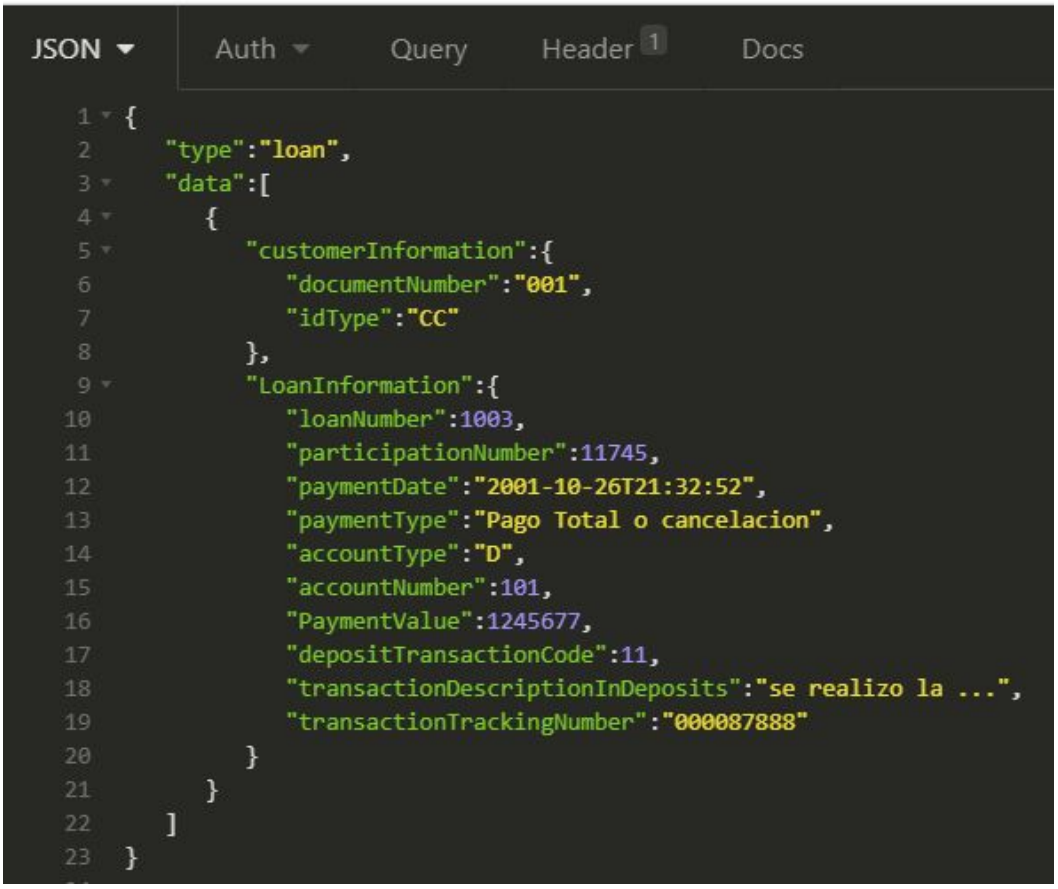
Código 30. Ejecución 'EndPoint HTTP' del Microservice.

6.1.2. Se ejecuta la instancia de Neo4j, puede ser la versión Docker o la versión desktop. En este caso se usa la versión desktop para una mejor visualización de los resultados.

6.1.3. Con el apoyo de un software capaz de realizar peticiones HTTP, se simula el evento en el componente 'Producer' e inmediatamente este envía el mensaje al bróker de mensajería (ver Figuras 17, 18 y 19). En este caso es un préstamo:

POST http://127.0.0.1:9090/event

JSON Auth Query Header 1 Docs

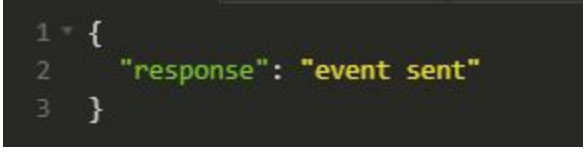


```
{
   "type":"loan",
   "data":[
      {
         "customerInformation":{
            "documentNumber":"001",
            "idType":"CC"
         },
         "LoanInformation":{
            "loanNumber":1003,
            "participationNumber":11745,
            "paymentDate":"2001-10-26T21:32:52",
            "paymentType":"Pago Total o cancelacion",
            "accountType":"D",
            "accountNumber":101,
            "PaymentValue":1245677,
            "depositTransactionCode":11,
            "transactionDescriptionInDeposits":"se realizo la ...",
            "transactionTrackingNumber":"000087888"
         }
      }
   ]
}
```

Figura 17. Creación de un evento 'Loan' usando el API del 'Producer'.

200 OK 22.1 ms 31 B

Preview Header 4 Cookie



```
{
   "response": "event sent"
}
```

Figura 18. Respuesta petición de la Figura 17.

```
[x][store] Client node name: 001 , does not exist
[*][store] Client node name: 001 , was created
[x][store] BankingService node name: loan , does not exist
[*][store] BankingService node name: loan , was created
[*][store] Relationship (001)-USE->(loan) , was created
[*][store] Loan , was created
```

Figura 19. Logs producidos por el Microservice.

6.1.4. De manera análoga al paso anterior, se generan otros eventos:

- El usuario '001' desde la cuenta bancaria '101' realiza dos transferencias a la cuenta bancaria '102' del usuario '002'
- El usuario '002' desde la cuenta bancaria '102' realiza una transferencia a la cuenta bancaria '101' del usuario '001'

6.1.5. Usando el browser de Neo4j se visualizan los registros en la base de datos, resultados en la Figura 20.

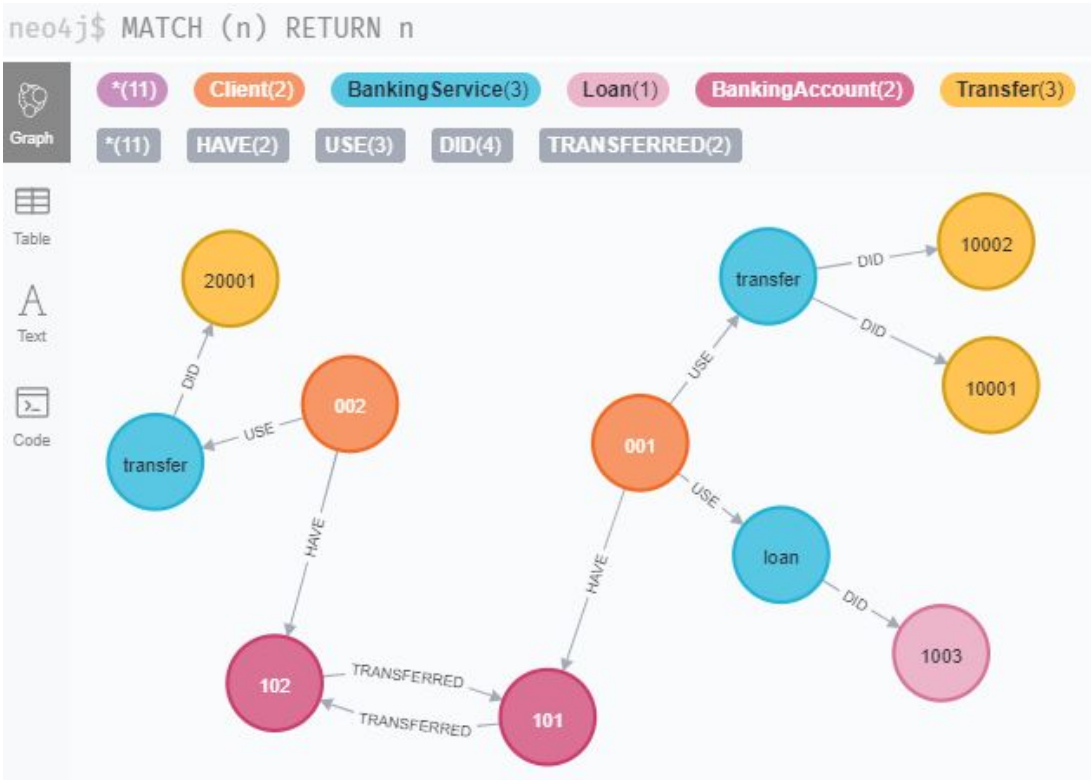


Figura 20. Registro exitoso de los eventos en la base de datos.

6.1.6. Con el software usado en el paso 3, se realizan dos peticiones al API HTTP del Microservice.

- Consultar las funcionalidades más usadas por el cliente (ver Figuras 21 y 22):

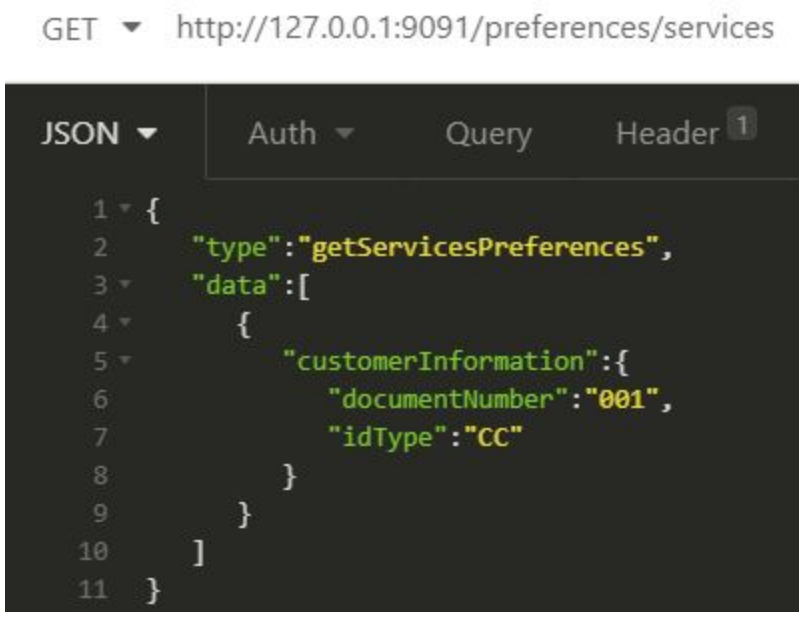


Figura 21. Petición para obtener las funcionalidades más usadas por el usuario.

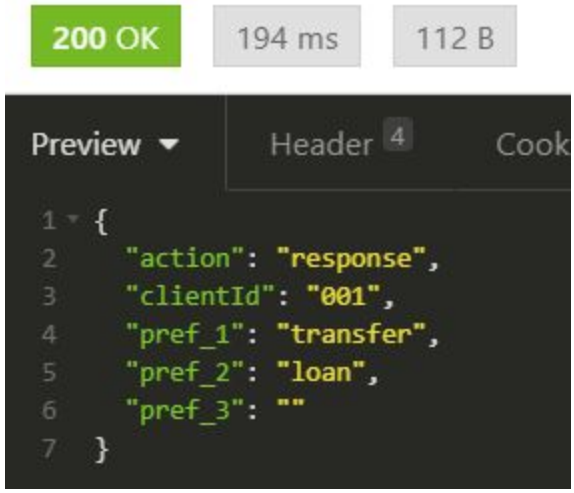


Figura 22. Respuesta a la petición de la Figura 21.

- Consultar las cuentas bancarias a las que más les transfiere una cuenta bancaria (Figuras 23 y 24):

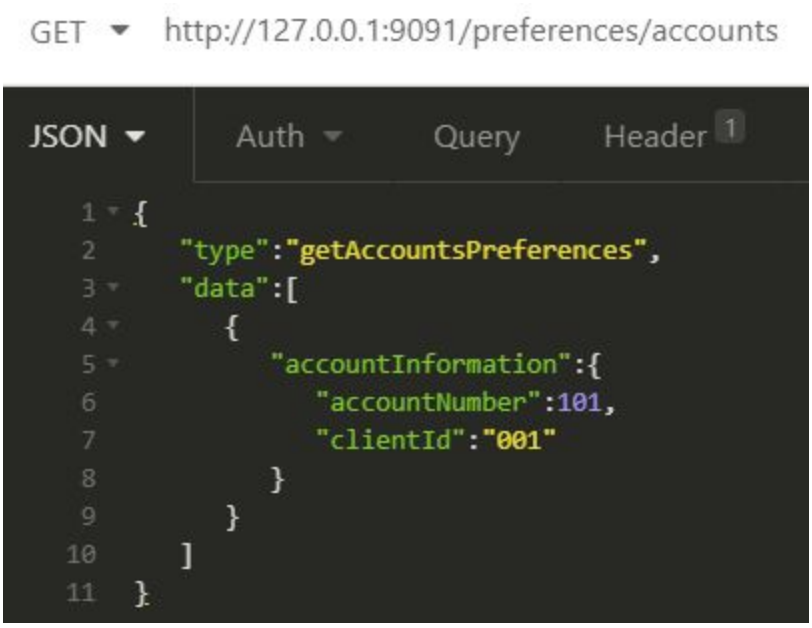


Figura 23. Petición para obtener las cuentas bancarias a las que más les transfiere una cuenta bancaria.

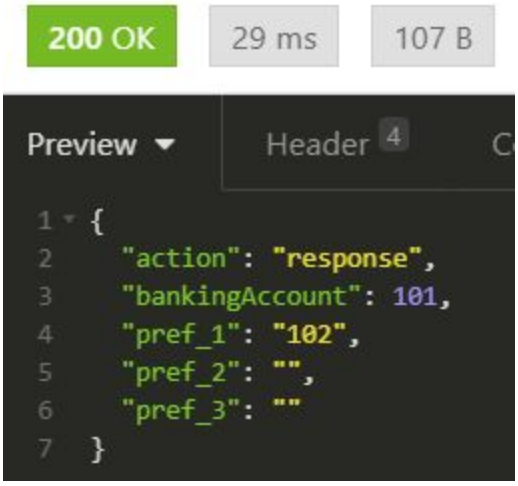


Figura 24. Respuesta a la petición de la Figura 23.

6.2. Repositorio

La solución se encuentra en un repositorio público:

https://github.com/DanielOrtizMontoya/UserPreferences

7. Conclusiones

- Se logró desarrollar e implementar un microservicio para reconocer las preferencias de los clientes en los canales digitales de Bancolombia.
- El microservicio también permite registrar información relacionada con el comportamiento de los clientes (interacciones), esta información es muy valiosa para futuros agentes.
- Los resultados obtenidos con este proyecto permitirán mejorar la experiencia del usuario de los clientes de Bancolombia cuando estos utilicen los canales digitales.
- Se adquirieron conocimientos sobre cómo abordar y solucionar problemas tecnológicos en un entorno corporativo.

8. Referencias Bibliográficas

[1] Chris Richardson (2018), Pattern: Microservice Architecture . [Internet] disponible en: https://microservices.io/patterns/microservices.html

[2] Armin Balalaie, Abbas Heydarnoori & Pooyan Jamshidi, Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture .

[3] Justin J. Miller (2013), Graph Database Applications and Concepts with Neo4 j. Proceedings of the Southern Association for Information Systems Conference, Georgia Southern University, Atlanta, GA, USA.

[4] Guía oficial de Neo4j para desarrolladores [Internet] Disponible en: https://neo4j.com/developer/graph-database/

[5] What is a Container? [Internet] Docker Inc. Disponible en: https://www.docker.com/resources/what-container

[6] RabbitMQ [Internet] VMware Inc. Disponible en: https://www.rabbitmq.com/

[7] The Py2neo Handbook [Interner]. Disponible en: https://py2neo.org/2020.1/